IELR(1): Practical LR(1) Parser Tables for Non-LR(1) Grammars with Conflict Resolution

Joel E. Denny and Brian A. Malloy
School of Computing
Clemson University
Clemson, SC 29634, USA
{jdenny,malloy}@cs.clemson.edu

ABSTRACT

There has been a recent effort in the literature to reconsider grammar-dependent software development from an engineering point of view. As part of that effort, we examine a deficiency in the state of the art of practical LR parser table generation. Specifically, LALR sometimes generates parser tables that do not accept the full language that the grammar developer expects, but canonical LR is too inefficient to be practical. In response, many researchers have attempted to develop minimal LR parser table generation algorithms. In this paper, we demonstrate that a well known algorithm described by David Pager and implemented in Menhir, the most robust minimal LR(1) implementation we have discovered, does not always achieve the full power of canonical LR(1) when the given grammar is non-LR(1) coupled with a specification for resolving conflicts. We also outline an original minimal LR(1) algorithm, IELR(1), which we have implemented as an extension of GNU Bison and which does not exhibit this deficiency. Finally, using our implementation, we demonstrate the relevance of this deficiency for several real-world grammars, and we show that our implementation is feasible for generating minimal LR(1) parsers for those grammars.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*parsing*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*syntax*; D.2.m [**Software Engineering**]: Miscellaneous

General Terms

Algorithms, Languages, Theory

Keywords

grammars, grammarware, LALR, LR, Yacc

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil
Copyright 2008 ACM 978-1-59593-753-7/08/0003 ...$5.00.

1. INTRODUCTION

Grammar-dependent software is omnipresent in software development [18]. For example, compilers, document processors, browsers, import/export tools, and generative programming tools are used in software development in all phases. These phases include comprehension, analysis, maintenance, reverse-engineering, code manipulation, and visualization of the application program under study. However, construction of these tools relies on the correct recognition of the language constructs specified by the grammar.

Some aspects of grammar engineering are reasonably well understood. For example, the study of grammars as definitions of formal languages, including the study of LL, LR, LALR, and SLR algorithms and the Chomsky hierarchy, form an essential part of most computer science curricula. Nevertheless, parsing as a disciplined study must be reconsidered from an engineering point of view [18, 19]. Many parser developers eschew the use of parser generators because it is too difficult to customize the generated parser or because the generated parser requires considerable modification to incorporate sufficient power to handle modern grammars such as the C++ and C# grammars. Thus, industrial strength parser development requires considerable effort, and many approaches to parser generation are ad hoc [26, 27].

From an engineering point of view, one source of difficulty in parser development stems from a deficiency in the state of the art of practical LR parser table generation. LR parsing is "the most general nonbacktracking shift-reduce parsing method known", and canonical LR is the most general technique for generating LR parser tables from a given grammar [12]. As a result, canonical LR parser tables accept the language that a grammar developer expects a given grammar to define. Unfortunately, canonical LR tables require "too much space and time to be useful in practice" [12]. In contrast, LALR parser tables are feasible and are thus employed by widely used tools like Yacc [17, 11] and its GNU implementation Bison [1]. Unfortunately, as Bison's manual points out, LALR parser tables contain "mysterious conflicts" that "don't look warranted" [16]. These conflicts cause the parser to encounter "unnatural errors" because LALR "is not powerful enough to match the intuition of the grammar writer" [24, 25]. In this way, LALR can worsen the difficulty of developing a correct grammar. Even for an existing correct LALR grammar, LALR can interfere with incremental changes [24, 25], which are inevitable in the face

1. $S \rightarrow aAa$
2. $\quad \rightarrow bAb$
3. $A \rightarrow a$
4. $\quad \rightarrow aa$

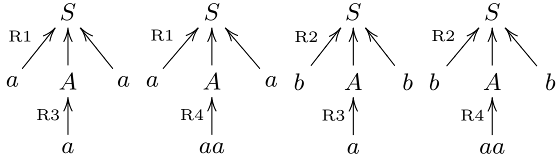

Figure 1: A Non-LR(1) Grammar Example. This grammar defines a language consisting of 4 sentences, each of which corresponds to 1 parse tree: *aaa*, *aaaa*, *bab*, and *baab*. Assuming that the parser is LR(1) and that *a* is left-associative, the sentence *aaaa* is dropped from the language.

Canonical LR(1)				LALR(1)			
0.	$S' \rightarrow \cdot S,$	$\{\lambda\}$	G1	0.	$S' \rightarrow \cdot S,$	$\{\lambda\}$	G1
	$S \rightarrow \cdot aAa,$	$\{\lambda\}$	S2		$S \rightarrow \cdot aAa,$	$\{\lambda\}$	S2
	$\rightarrow \cdot bAb,$	$\{\lambda\}$	S3		$\rightarrow \cdot bAb,$	$\{\lambda\}$	S3
1.	$S' \rightarrow S \cdot,$	$\{\lambda\}$	Acc	1.	$S' \rightarrow S \cdot,$	$\{\lambda\}$	Acc
2.	$S \rightarrow a \cdot Aa,$	$\{\lambda\}$	G4	2.	$S \rightarrow a \cdot Aa,$	$\{\lambda\}$	G4
	$A \rightarrow \cdot a,$	$\{a\}$	S8		$A \rightarrow \cdot a,$	$\{a\}$	S8
	$\rightarrow \cdot aa,$	$\{a\}$	S8		$\rightarrow \cdot aa,$	$\{a\}$	S8
3.	$S \rightarrow b \cdot Ab,$	$\{\lambda\}$	G5	3.	$S \rightarrow b \cdot Ab,$	$\{\lambda\}$	G5
	$A \rightarrow \cdot a,$	$\{b\}$	**S10**		$A \rightarrow \cdot a,$	$\{b\}$	**S8**
	$\rightarrow \cdot aa,$	$\{b\}$	**S10**		$\rightarrow \cdot aa,$	$\{b\}$	**S8**
4.	$S \rightarrow aA \cdot a,$	$\{\lambda\}$	S6	4.	$S \rightarrow aA \cdot a,$	$\{\lambda\}$	S6
5.	$S \rightarrow bA \cdot b,$	$\{\lambda\}$	S7	5.	$S \rightarrow bA \cdot b,$	$\{\lambda\}$	S7
6.	$S \rightarrow aAa \cdot,$	$\{\lambda\}$	R1	6.	$S \rightarrow aAa \cdot,$	$\{\lambda\}$	R1
7.	$S \rightarrow bAb \cdot,$	$\{\lambda\}$	R2	7.	$S \rightarrow bAb \cdot,$	$\{\lambda\}$	R2
*8.	$A \rightarrow a \cdot,$	$\boldsymbol{\{a\}}$	R3	*8.	$A \rightarrow a \cdot,$	$\boldsymbol{\{ab\}}$	R3
	$A \rightarrow a \cdot a,$	~~$\boldsymbol{\{a\}}$~~	~~S9~~		$A \rightarrow a \cdot a,$	~~$\boldsymbol{\{ab\}}$~~	~~S9~~
9.	$A \rightarrow aa \cdot,$	$\boldsymbol{\{a\}}$	R4	9.	$A \rightarrow aa \cdot,$	$\boldsymbol{\{ab\}}$	R4
10.	$\boldsymbol{A \rightarrow a \cdot,}$	$\boldsymbol{\{b\}}$	**R3**				
	$\boldsymbol{A \rightarrow a \cdot a,}$	$\boldsymbol{\{b\}}$	**S11**				
11.	$\boldsymbol{A \rightarrow aa \cdot,}$	$\boldsymbol{\{b\}}$	**R4**				

Table 1: Example Parser Tables. These are the canonical LR(1) and LALR(1) parser tables for the grammar of Figure 1. λ in a lookahead set represents the end of the input. Differences between canonical LR(1) and LALR(1) are shown in bold. In both cases, state 8 has a S/R conflict on *a* resolved as a reduce since *a* is left-associative.

of software evolution.

In response, many researchers have developed *minimal LR* algorithms, which attempt to generate parser tables with the power of canonical LR but with nearly the efficiency of LALR [20, 22, 21, 24, 25, 23, 8, 6]. Menhir, hosted at [7], is an implementation of Pager's algorithm, which is described in [21]. Menhir is the most robust minimal LR(1) implementation we have discovered available.

In this paper, we show that Pager's algorithm and thus Menhir are not always able to generate parser tables with the full power of canonical LR(1) if the given grammar is non-LR(1) coupled with a specification for resolving conflicts. We also describe an original minimal LR(1) algorithm, IELR(1), that does not suffer from this deficiency. In section 2, we present an original non-LR(1) grammar example to demonstrate the deficiency of Pager's algorithm. We also introduce some original terminology to facilitate further discussion. In section 3, we outline our IELR(1) algorithm, which we have implemented as an extension of Bison. In section 4, we compare the Bison LALR(1) implementation with our IELR(1) implementation using our example grammar plus five grammars for popular languages as case studies. In section 5, we review related work. In section 6, we use the results of our case studies to conclude that (1) Menhir's deficiency does affect real-world parsers, (2) it can create bugs relative to the intended design of such parsers, and (3) IELR(1) is feasible for generating minimal LR(1) parsers for sophisticated real-world grammars.

2. MOTIVATING EXAMPLE

In this section, we demonstrate an original non-LR(1) grammar example. We analyze the language defined by that grammar in section 2.1. In sections 2.2 and 2.3, we describe its canonical LR(1) and LALR(1) parser tables in order to introduce some original terminology to facilitate discussion. In section 2.4, we identify a deficiency of Pager's algorithm and thus of Menhir.

2.1 A Non-LR(1) Grammar

As written, the grammar in Figure 1 defines a language consisting of 4 sentences: *aaa*, *aaaa*, *bab*, and *baab*. Consider how an LR parser behaves when it reaches the $\cdot$ in either of the marked input sentences, $ba \cdot b$ and $ba \cdot ab$. These sentences look the same before the $\cdot$ and so the parser performs identical actions until this point. If the parser then looks ahead 1 token and sees *b*, it knows it must reduce the previous *a* to *A* as in the third parse tree in Figure 1. If it sees *a* instead, it must shift the *a* so that it can then reduce the previous *aa* to *A* as in the fourth parse tree. Similarly, at the $\cdot$ in $aa \cdot a$ and $aa \cdot aa$, the parser must choose to reduce in order to accept *aaa* as in the first parse tree, but it must choose to shift in order to accept *aaaa* as in the second parse tree. However, in this case, the first token of lookahead is the same, *a*, and so does not distinguish between the possible parser actions.

Assume that the grammar author has designed the grammar with LR(1) in mind. If he has declared *a* as left-associative, the parser chooses to reduce in the cases of both $aa \cdot a$ and $aa \cdot aa$. In this way, the grammar author has specified a new language consisting of only 3 sentences: *aaa*, *bab*, and *baab*. Notice how we seem to be able to determine the language without computing any parser tables.

2.2 New Canonical LR(1) Terminology

The first column of Table 1 shows the canonical LR(1) parser tables for the grammar of Figure 1 when *a* is left-associative. We introduce the term *isocores* to describe states that have the same core. For example, state 8 is an isocore of state 10, but their lookahead sets and actions are different. When we speak of a particular state's set of viable prefixes, we are speaking of the set of possible viable prefixes that could be on the stack when that state is at the top of the stack. For example, state 8's only viable prefix is

Canonical LR(1)			LALR(1)		
Stack	Input	Action	Stack	Input	Action
0	*baab*	S3	0	*baab*	S3
0,3	*aab*	S10	0,3	*aab*	S8
0,3,10	*ab*	S11	0,3,8	*ab*	R3
0,3,10,11	*b*	R4	0,3	*Aab*	G5
0,3	*Ab*	G5	0,3,5	*ab*	Err
0,3,5	*b*	S7			
0,3,5,7		R2			
0	*S*	G1			
0,1		Acc			

Table 2: An Example Parse. This table shows how the canonical LR(1) and LALR(1) parser tables of Table 1 parse the sentence *baab*. Because LALR(1) merges state 10 into state 8, it rejects the sentence even though the canonical LR(1) parser accepts it.

---

aa, and state 10's only viable prefix is *ba*. State 8 contains the S/R conflict on token *a* whose resolution as reduce (1) permits a successful completion of the parse of $aa \cdot a$ but (2) leads the parse of $aa \cdot aa$ to a syntax error. We say the shift and reduce actions are *contributions* to the conflict. We call the reduce action the *dominant contribution*.

2.3 LALR(1) Conflicts Reconsidered

The second column of Table 1 shows the LALR(1) parser tables for the grammar in Figure 1. They are nearly identical to the canonical LR(1) parser tables but isocores are merged together. For example, canonical LR(1) state 10 is merged into state 8. Merging isocores can create new conflicts that are not present in the canonical LR(1) parser tables. [16] calls these *mysterious conflicts* because they can be a source of confusion for grammar authors. The trouble is that a simple grammar analysis as we performed in section 2.1 does not easily reveal mysterious conflicts. To discover them, the grammar author is forced to examine parser tables. In this paper, we rename [16]'s mysterious conflicts to *mysterious new conflicts* since we wish to emphasize that this term always refers to conflicts that do not exist in the canonical LR(1) parser tables. As [16] points out, such conflicts are always R/R conflicts since, as [12] points out, merging isocores can never create new S/R conflicts.

Because canonical LR(1) states 10 and 8 in Table 1 are merged, the LALR(1) parser encounters the conflict of state 8 even after the viable prefix of state 10. Unfortunately for $ba{\cdot}ab$, which starts with that viable prefix and which requires a shift in state 10, the conflict is resolved as a reduce. As Table 2 demonstrates, this leads the LALR(1) parse of $ba \cdot ab$ to a syntax error even though canonical LR(1) is able to parse it successfully. The conflict here is not a mysterious new conflict since it already exists in canonical LR(1) state 8. Instead, we have identified a second category of mysterious conflicts that we call *mysterious invasive conflicts*. That is, switching from canonical LR(1) to LALR(1) can cause a parser to encounter existing conflicts after viable prefixes for which it never would have encountered those conflicts before and, as a result, to perform incorrect actions. This switch can also cause a parser to perform incorrect actions after viable prefixes for which it would have encountered the existing conflict before. We call this a *mysterious mutated conflict*. Since mysterious conflicts of these two categories stem from existing conflicts in canonical LR(1) tables, they are only possible for non-LR(1) grammars.

In summary, given the grammar of Figure 1, given that *a* is left-associative, and assuming an LR(1) parser, the language so specified is a set of 3 sentences: *aaa*, *bab*, and *baab*. As expected, the canonical LR(1) parser tables for this specification accept exactly that language. However, LALR(1) diminishes the language to only 2 sentences: *aaa* and *bab*. The culprit is not a mysterious new conflict. The culprit is a mysterious invasive conflict.

2.4 The Deficiency of Pager's Algorithm

Most algorithms for generating LR(1) parser tables merge states only if they pass some sort of compatibility test. Canonical LR(1) parser tables are relatively large because the compatibility test is relatively restrictive: states must be isocores and their lookahead sets must be identical. LALR(1) parser tables are sometimes too small to maintain the full power of LR(1) because the compatibility test is too lenient: states must simply be isocores. [21] describes an algorithm to generate LR(1) parser tables while employing tests for two kinds of compatibility, weak compatibility and strong compatibility, both of which lie somewhere in between. Throughout this paper, we refer to this algorithm as *Pager's algorithm*.

We identify two problems with Pager's algorithm for our purposes. First, its weak compatibility test is not designed for non-LR(1) grammars. Thus, the only category of mysterious conflicts that the weak compatibility test can always avoid is mysterious new conflicts. For example, it does not avoid the mysterious invasive conflict in Table 1. Second, both its weak and strong compatibility tests permit isocores to be merged if the merged state would not contain a potential conflict, but the definition of potential conflict employed does not include potential or even real S/R conflicts. Thus, even using the strong compatibility test by itself, Pager's algorithm does not avoid the mysterious invasive conflict in Table 1.

Like Bison, Menhir is designed to accept non-LR(1) grammars with a specification for resolving conflicts. Thus, even though Menhir employs Pager's algorithm to try to achieve the power of canonical LR(1), it fails to in some cases. For example, we have confirmed that, when given the grammar of Figure 1 combined with the declaration of *a* as left-associative, Menhir generates a parser that does not accept the sentence *baab*. Again, a canonical LR(1) parser does accept this sentence.

3. METHODOLOGY

In this section, we outline our IELR(1) algorithm in 6 phases. We label these phases Phase 0 through Phase 5 and describe them in sections 3.1 through 3.6. Due to space restrictions, we omit the details. For a complete description, see our technical report at [14].

3.1 Phase 0: LALR(1)

Pager's algorithm *avoids* some mysterious conflicts by refusing to merge isocores if its compatibility tests *predict* that doing so would induce an inadequacy in the parser tables. In contrast, IELR(1) computes up front all possible inadequacies, and then afterwards it *eliminates* all the inadequacies that would not appear in canonical LR(1) parser tables. Since LALR(1) parser tables have all isocores merged fully,

they contain all such inadequacies. Thus, Phase 0 computes LALR(1) parser tables. It does so in two steps: (1) compute LR(0) parser tables, and (2) compute the reduction lookahead sets using the technique described by [15]. As part of its algorithm, step 2 also computes a set of goto tables that the remaining IELR(1) phases require. In our implementation of Phase 0, we use Bison's existing LALR(1) implementation, which already computes those goto tables.

3.2 Phase 1: Compute Auxiliary Tables

The remaining phases of IELR(1) require three additional tables beyond those computed in Phase 0, and they are computed in Phase 1. $follow_kernel_items$ records dependencies of goto follow sets on kernel item lookahead sets. $always_follows$ records goto follow tokens that do not depend on the kernel item lookahead sets of predecessor states. That is, this table records goto follows that will never change no matter how IELR(1) may split the LALR(1) parse states. $predecessors$ records transition predecessor relations between LALR(1) states.

3.3 Phase 2: Compute Annotations

Phase 2 annotates inadequate LALR(1) states. Each annotation describes how isocores of the annotated state might contribute to an inadequacy. The $predecessors$ table enables reverse iteration from the conflicted state to all the predecessor states that might contribute to the same inadequacy. This iteration is similar to lane tracing as described by [20] and [22]. The $follow_kernel_items$ table, $always_follows$ table, and goto tables enable the computation of the inadequacy contributions made by each state.

3.4 Phase 3: Split States

Phase 3 recomputes the parse states in a manner similar to Phase 0 step 1. Whereas Phase 0 step 1 merges all isocores together, Phase 3 only merges isocores if they will maintain the same dominant contribution to every inadequacy according to their core's annotations. In order to do this, Phase 3 must compute partial kernel item lookahead sets by using the $follow_kernel_items$ and $always_follows$ tables and by propagating lookaheads from each parse state to its successor if the lookaheads appear in the annotations.

3.5 Phase 4: Compute Reduction Lookaheads

Phase 4 runs step 2 of Phase 0 again without modification. That is, it computes the full lookahead sets on reductions in all IELR(1) parse states.

3.6 Phase 5: Resolve Remaining Conflicts

All that's left is to resolve the remaining conflicts in the parser tables. Our IELR(1) implementation uses Bison's existing conflict resolution algorithm without modification.

4. RESULTS OF USING IELR(1)

In this section, we compare the Bison LALR(1) implementation with our IELR(1) implementation using six grammars as case studies, which are summarized in Table 3. In section 4.1, we describe the grammars in detail. In section 4.2, we compare the parser tables that the algorithms generate for the grammars. In section 4.3, we compare the performance of the algorithms in terms of time and space.

4.1 Case Studies

Grammar	Version	$\|T\|$	$\|V\|$	$\|T \cup V\|$	$\|P\|$
Figure 1		2	2	4	4
Gawk	Gawk 3.1.0	61	45	106	163
Gpic	Groff 1.18.1	138	45	183	247
C	GCC 4.0.4	92	208	300	573
Java	GCC 4.2.1	109	164	273	516
C++	ISO 2003	117	184	301	481

Table 3: Grammar Characteristics. These counts measure the size of each case study's grammar $G = (V, T, P, S)$, such that V is the set of nonterminals, T is the set of terminals or tokens, P is the set of productions, and S is the start symbol. These counts include the productions and nonterminals that Bison generates implicitly for mid-rule actions.

Table 3 characterizes the grammars of our case studies. Our first case study is the example grammar of Figure 1 including the declaration of a as left-associative.

Our next four case studies are mature grammars from widely used software products employing LALR(1) parser generators. Gawk (GNU AWK), a text-based data processing language, was first written in 1986 but is based on the original AWK, which was written in 1977 and which is standardized in SUSv3 (the Single UNIX Specification, Version 3) [11, 2]. Groff (GNU Troff) is a document formatting system for UNIX that includes Gpic (GNU Pic), a Groff preprocessor for specifying diagrams. Groff was first released in 1990 and is based on Troff which has existed since the early 1970's [13, 4]. We copied our C and Java grammars from GCC (the GNU Compiler Collection), which is a widely used collection of compilers developed by the GNU Project [3].

The latest version of the C++ programming language is C++ 2003, the formal specification for which is [10]. Annex A of that specification presents a formal C++ grammar. As our final case study, we formatted this grammar as a Bison grammar file except that, for section A.2, Lexical conventions, we replaced the $integer_literal$, $character_literal$, $floating_literal$, and $string_literal$ nonterminals with tokens.

4.2 LALR(1) vs. IELR(1) Parser Tables

Table 4 describes the parser tables that the Bison LALR(1) implementation and our IELR(1) implementation generate for each of our case studies' grammars. Since some grammars include token precedence and associativity declarations that resolve most of their conflicts, we also describe their parser tables when generated without these declarations in order to better demonstrate grammar analysis complexity.

When IELR(1) splits states into isocores, each conflict from the original LALR(1) state might be duplicated among several isocores, and Bison then counts the conflict separately for each. Sometimes the multiple count is a misleading representation of grammar complexity because all isocores have perfect duplicates of the conflict and thus the same precedence and associativity declarations can resolve them all. Therefore, from each IELR(1) unresolved conflict count in Table 4, we subtract all but one copy of each unresolved conflict that is perfectly duplicated among isocores.

Table 5 describes the parser actions that are corrected in the parser tables by switching from LALR(1) to IELR(1).

Grammar	States		S/R		R/R	
	LA	IE	LA	IE	LA	IE
Figure 1	10	12	0	0-0	0	0-0
no prec/assoc	11	11	1	1-0	0	0-0
Gawk	320	329	65	65-0	0	0-0
no prec/assoc	320	320	410	410-0	0	0-0
Gpic	423	428	0	0-0	0	0-0
no prec/assoc	426	426	803	803-0	8	8-0
C	933	933	13	13-0	0	0-0
no prec/assoc	933	933	329	329-0	0	0-0
Java	792	792	0	0-0	62	62-0
C++	822	836	407	410-3	135	169-34

Table 4: Parser Tables. In this table, we report the number of states, the number of unresolved S/R conflicts, and the number of unresolved R/R conflicts in the parser tables that the Bison LALR(1) implementation and our IELR(1) implementation generate for our case studies' grammars. For IELR(1) conflict counts, we show adjustments to account for unresolved conflicts that are perfectly duplicated among isocores.

Grammar	Actions	States	Tokens
Figure 1	1	1	1
Gawk	9	3	3
Gpic	2	1	2
C	0	0	0
Java	0	0	0
C++	4	4	2

Table 5: Action Corrections. For each of our case studies' grammars, this table reports the number of parser actions that are corrected by switching from LALR(1) to IELR(1), the number of parser states containing corrected parser actions, and the number of unique tokens in the grammar on which there are corrected parser actions.

For the Figure 1, Gawk, and Gpic grammars, every corrected action originates from a mysterious invasive LALR(1) S/R conflict resolved as a reduce action, and only the shift action remains in each of the new corrected IELR(1) isocores.

In the case of the Figure 1 grammar, the LALR(1) and IELR(1) parser tables generated by Bison are nearly the same as the LALR(1) and canonical LR(1) parser tables shown in Table 1. That is, when a is declared left-associative, LALR(1) accepts only 2 input sentences, but IELR(1) accepts the same set of 3 input sentences that canonical LR(1) does.

After exploring the Gpic grammar's source comments, we conclude that IELR(1) corrects the behavior of a Gpic feature that was intentionally created by the author of Gpic's grammar. However, we also contacted the current Gpic developers for their opinion. They seem to have been previously unaware that the affected feature even exists, and some members expressed an interest in seeing it removed. For the full discussion, see [5].

We have confirmed that Menhir is unable to recognize the need to split off any corrected isocore for the Figure 1, Gawk, or Gpic grammar and so leaves the incorrect actions.

Grammar	Real Run Time (s)		Peak Mem Usage (KB)		Annotations
	LA	IE	LA	IE	
Figure 1	0.038	0.038	60	62	1
Gawk	0.065	0.084	110	195	2537
Gpic	0.117	0.162	115	450	6123
C	0.118	0.150	225	650	6573
Java	0.149	0.250	375	900	4636
C++	0.080	0.259	425	975	5346

Table 6: Parser Tables Computation. This table describes the performance of the Bison LALR(1) implementation and our IELR(1) implementation for each of our case studies' grammars. The real run time measures the full run time for Bison. However, we report peak memory usage only for the duration of the LALR(1) or IELR(1) algorithm. We also report the total number of inadequacy annotations attached to the LALR(1) states during phase 2 of IELR(1).

In this way, LALR(1) and Menhir fail to generate parsers that accurately implement these grammars, but IELR(1) is successful.

4.3 LALR(1) vs. IELR(1) Performance

In Table 6, we report the performance of the Bison LALR(1) implementation and our IELR(1) implementation for each of our case studies' grammars. We performed these measurements on a Toshiba Tecra M4-S435 with an Intel Pentium M Processor 740 [1.73GHz, 2MB L2, 533MHz FSB] with 1024MB DDR2 SDRAM running the Slackware 10.2.0 distribution of GNU/Linux with Linux kernel version 2.6.13. Using GNU time 1.7, we measured the full real run time of Bison, and we report the average for 3 trial runs. We estimated peak memory usage during the execution of the LALR(1) and IELR(1) algorithms based on diagrams generated by Massif from Valgrind 3.2.3 [9].

5. RELATED WORK

[18, 19, 26, 27] discuss the need to improve the state of the art in grammar-dependent software engineering. [20, 22, 21, 24, 25, 23, 8, 6] develop minimal LR algorithms. Menhir, hosted at [7], is an implementation of Pager's algorithm, which is described in [21]. It is the most robust minimal LR(1) implementation we have discovered available, but it is not always able to generate parser tables with the full power of canonical LR(1) if the given grammar is non-LR(1) coupled with a specification for resolving conflicts.

6. CONCLUDING REMARKS

We are surprised to discover that mature grammars from widely used software products employing LALR(1) parser generators should suffer from any incorrect parser actions that result from the misuse of the LALR(1) algorithm. The Gawk and Gpic case studies provide strong evidence that such incorrect actions do occur in real-world parsers. Such actions are unintuitive and thus may impede the development of a correct parser. Moreover, the Gpic case study shows that such incorrect actions can create actual bugs relative to the intended design of a real-world LALR(1)-generated parser. Pager's algorithm and thus Menhir do not

address the incorrect actions in either of these case studies. IELR(1) corrects them for both.

Because of the maturity of the GCC project, we are not surprised that its C and Java grammars suffer from no incorrect parser actions that result from a misuse of the LALR(1) algorithm. These case studies demonstrate IELR(1)'s ability to recognize when LALR(1) parser tables are sufficient without unnecessarily splitting its states into additional canonical LR(1) states.

The performance measurements from all of our case studies show that IELR(1) is feasible for generating minimal LR(1) parsers for sophisticated real-world grammars. Specifically, for each of our case studies when using the IELR(1) algorithm, Bison did not run longer than 0.3 seconds, and the IELR(1) algorithm never required as much as 1 MB of memory at any point in time.

7. REFERENCES

[1] Bison. The GNU Project. `http://www.gnu.org/software/bison/`.

[2] Gawk. The GNU Project. `http://www.gnu.org/software/gawk/`.

[3] GCC. The GNU Project. `http://gcc.gnu.org/`.

[4] Groff. The GNU Project. `http://www.gnu.org/software/groff/`.

[5] [Groff] parsing a corner specification. Groff mailing list archives. `http://lists.gnu.org/archive/html/groff/2007-08/msg00051.html`.

[6] LRGen. P. B. Mann. http://www.paulbmann.com/lrgen/.

[7] Menhir. F. Pottier and Y. Régis-Gianas. `http://cristal.inria.fr/~fpottier/menhir/`.

[8] The Honalee LR(k) Algorithm. D. R. Tribble. `http://david.tribble.com/text/honalee.html`.

[9] Valgrind. `http://www.valgrind.org/`.

[10] *International Standard, Programming Languages – C++*. Number ISO/IEC 14882:2003(E). American National Standards Institute, October 2003.

[11] Single UNIX Specification, Version 3. The IEEE and the Open Group, April 2004. `http://www.opengroup.org/bookstore/catalog/t041.htm`.

[12] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[13] Ralph Corderoy. troff.org, The Text Processor for Typesetters. `http://troff.org`.

[14] Joel E. Denny. `http://joeldenny.org`.

[15] F. DeRemer and T. Pennello. Efficient Computation of LALR(1) Look-Ahead Sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, October 1982.

[16] The GNU Project. *Bison*, 2.3 edition, May 2006.

[17] S. Johnson. Yacc: Yet Another Compiler Compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[18] P. Klint, R. Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

[19] R. Lämmel. Grammar Testing. In *FASE*, pages 201–216, 2001.

[20] D. Pager. The lane tracing algorithm for constructing LR(k) parsers. In *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 172–181, New York, NY, USA, 1973. ACM Press.

[21] D. Pager. A Practical General Method for Constructing LR(k) Parsers. *Act Informatica*, 7(3):249–268, September 1977.

[22] D. Pager. The Lane-Tracing Algorithm for Constructing LR(k) Parsers and Ways of Enhancing Its Efficiency. *Information Sciences*, 12(1):19–42, 1977.

[23] P. Pepper. LR Parsing = Grammar Transformation + LL Parsing, Making LR Parsing More Understandable And More Efficient. Technical Report 99-05, TU Berlin, 1999.

[24] D. Spector. Full LR(1) parser generation. *SIGPLAN Not.*, 16(8):58–66, 1981.

[25] D. Spector. Efficient full LR(I) parser generation. *SIGPLAN Not.*, 23(12):143–150, 1988.

[26] M. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Trans. Softw. Eng. Methodol.*, 5(1):1–41, 1996.

[27] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Current Parsing Techniques in Software Renovation Considered Harmful. In *IWPC*, page 108, Washington, DC, USA, 1998.